(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 November 2004 (04.11.2004)

PCT

(10) International Publication Number
WO 2004/095457 A2

(51) International Patent Classification[7]: G11C

(21) International Application Number: PCT/US2004/011219

(22) International Filing Date: 12 April 2004 (12.04.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

10/412,045	11 April 2003 (11.04.2003)	US
10/411,835	11 April 2003 (11.04.2003)	US
10/411,784	11 April 2003 (11.04.2003)	US

(71) Applicant (*for all designated States except US*): **BITFONE COPRORATION** [US/US]; 32451 Golden Lantern, Suite 301, Laguna Niguel, CA 92677 (US).

(72) Inventors; and

(75) Inventors/Applicants (*for US only*): **O'NEILL, Patrick** [US/US]; 1 Antigua, Dana Point, CA 92629 (US). **SOTOS, Peter L.** [US/US]; 152 N. Singingwood Street, # 4, Orange, CA 92869 (US). **JACOBI, Sidney Andrew** [US/US]; 252 Muirfield Way, San Marcos, CA 92069 (US). **LIM, Jeong M.** [KR/US]; 101-1305 Chung-goo, Bluehill House, Jamwon-dong, Seocho-gu, Seoul (KR). **SHAO-CHUN, Chen** [—/US]; 27662 Aliso Creek Road, Apt. # 7304, Aliso Viejo, CA 92656 (US).

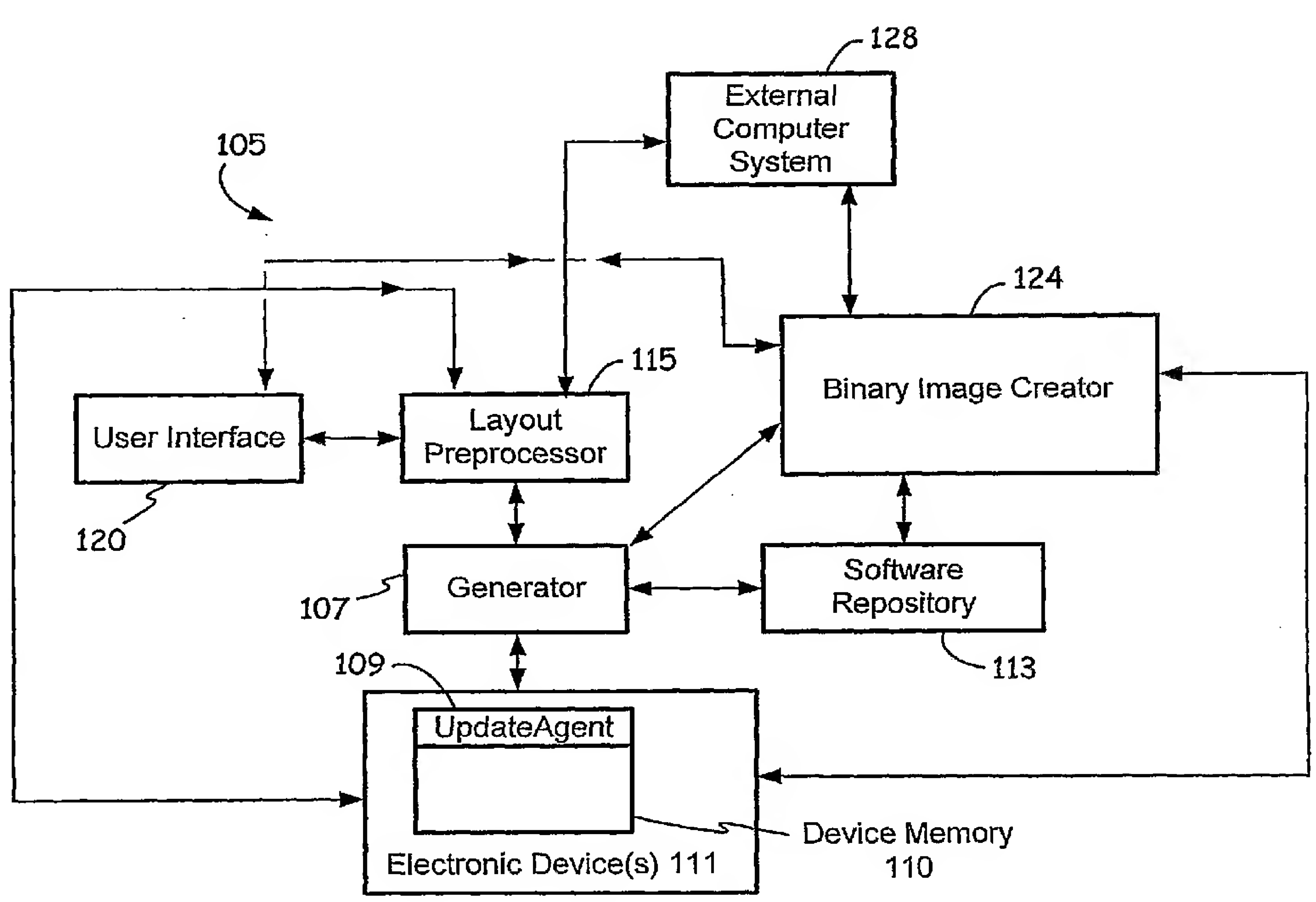
(74) Agent: **BORG, Kevin E.**; 500 West Madison Street, Suite 3400, Chicago, Illinois 60661 (US).

(81) **Designated States** (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) **Designated States** (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR,

[Continued on next page]

(54) Title: INITIALIZATION AND UPDATE OF SOFTWARE AND/OR FIRMWARE IN ELECTRONIC DEVICES



(57) **Abstract:** A system and method to effectively and efficiently update a version of firmware resident in a device memory is provided. A method of identifying one or more versions of firmware is provided by way of initializing a device memory with a known pattern. In addition, the amount of free unused memory space may be identified and calculated in a device memory. The system and method generates software update packages that are minimal in size providing cost benefit to a manufacturer and convenience to a user.

WO 2004/095457 A2

GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

INITIALIZATION AND UPDATE OF SOFTWARE AND/OR FIRMWARE IN ELECTRONIC DEVICES

RELATED APPLICATIONS

[01] This application makes reference to and claims priority based on the following provisional applications, the complete subject matter of each of which is incorporated herein by reference in its entirety.

Ser. No.	Docket No.	Title	Filed	Inventors
60/373,422	13888US01	Update Package Generation and Distribution Network	April 12, 2002	O'Neill Rao
60/373,421	13889US01	Pattern Detection Preprocessor in an Update Generation System	April 12, 2002	Chen O'Neill Rao Lilley
60/373,423	13890US01	Layout Preprocessor in an Update Generation System	April 12, 2002	Chen O'Neill
60/372,066	13922US01	Memory Initialization System for Initializing a Memory Image with a Pattern	April 12, 2002	Chen O'Neill

[02] This application is a continuation-in-part of U.S. Patent Application Ser. No. 10/311,462, "System and Method for Updating and Distributing Information", filed December 13, 2002, which is the National Stage filing of PCT Application Ser. No. PCT/US01/44034, "System and Method for Updating and Distributing Information", filed November 19, 2001, which claims priority to U.S. Provisional Patent Application Ser. No. 60/249,606, filed November 17, 2000, the complete subject matter of each of which is incorporated herein by reference in its entirety.

[03] This application is also related to the following co-pending applications, each of which is hereby incorporated herein by reference in its entirety:

Ser. No.	Docket No.	Title	Filed	Inventors
	13888US02	Update Package Generation and Distribution Network	April 11, 2003	O'Neill Rao
	13889US02	Pattern Detection Preprocessor in an Electronic Device Update Generation System	April 11, 2003	Chen O'Neill Rao Lilley

[04] federally sponsored research or development

[05] [Not Applicable]

[MICROFICHE/COPYRIGHT REFERENCE]

[06] [Not Applicable]

TECHNICAL FIELD

[07] The present invention relates generally to the process of updating software in electronic devices in general, and, more specifically, to the specification and subsequent execution of software update activities in such devices.

BACKGROUND OF THE INVENTION

[08] Electronic devices such as mobile phones, personal digital assistants, pagers, and interactive digital appliances have become indispensable as a tool as the number of features and functions they provide increases. As technology continues to evolve, a manufacturer of such devices will find it imperative to update these devices with revised software that enables a number of new features and functions.

[09] The term "software" used herein is intended to include not only software, but also or alternatively firmware. Similarly, the term "firmware" used herein is intended to include not only firmware, but also or alternatively software.

[10] When existing software in memory of an electronic device must be updated as a result of a version upgrade, the new version of the software may require more memory space than the existing version. An attempt to replace the existing version with the new version by replacement with new software may cause a shifting or relocation of adjacent code not related to the application software. Such relocations require the modification of software code in order to provide continued operation of the updated feature or function. These modifications often involve re-compiling any software code that is affected by address references to the relocated code. As a result, the size of an update may be quite large. The effects of shifting and relocation of adjacent software code as a part of a software update may result in significant costs to a manufacturer. Furthermore, the time it takes to update a software may increase due to a large update size, and this provides an inconvenience to any user.

[11] In order to perform a software update, it is important to evaluate the free memory space available should the update size occupy a larger space than the existing software. However, it is often difficult to assess the amount of free or unused memory available in an electronic device's memory because the free space may contain un-initialized random bit patterns. Free or unused memory in electronic devices provided by different manufacturers is usually not initialized, making it difficult to determine the location and size of any unused or free space. When performing updates of firmware in an electronic device, the amount of unused memory required for installation is often critical.

[12] During a manufacturing cycle, a manufacturer typically burns copies of identical software modules for their devices. However, the unused or free space may not have a consistent binary pattern. As a result, any two similar devices with the same make and model may often end up with different memory (ROM, RAM or FLASH) snapshots or images. When attempting to determine the difference between versions of software resident in the memory of any two devices, any unused space in either memory that contains random un-initialized sequences makes it difficult to compare the two.

[13] In the case of mobile handsets, the software that is often updated and stored in read-only memory (ROM) or programmable ROM (PROM) is called firmware. Firmware is often responsible for the behavior of a system when it is first switched on and is used to properly boot up a mobile handset when powered up. The unused memory space generated by a particular firmware image may be filled with a random sequence of binary data. This results in firmware images that are not necessarily consistent across handsets with the same firmware version. In comparing the firmware images from two different handsets of the same type, the firmware may appear to be significantly different from each other, when in reality they may be identical. As a result, it becomes a difficult task to easily identify the version of software loaded in an electronic device based on the image it presents. Further, it becomes difficult to determine whether a device requires an update.

[14] Further limitations and disadvantages of conventional and traditional approaches will become apparent to one of skill in the art, through comparison of such systems with some aspects of the present invention as set forth in the remainder of the present application with reference to the drawings.

BRIEF SUMMARY OF THE INVENTION

[15] A system is provided to effectively update a software and/or firmware in an electronic device. The update system comprises a memory layout preprocessor used in formulating a memory layout of the firmware, a user interface to provide input data from a user, a binary image creator used to generate a firmware initialization binary pattern, a generator to provide a desirable software update package for incorporation into a device memory, a software repository for storing binary image patterns and software modules and objects, an external computer system for storing memory layout specifications, an update agent used for processing the software update package generated by the generator, and an electronic device containing the device memory in which the firmware to be updated is resident.

[16] A method is provided for effectively updating a software and/or firmware in an electronic device. In one embodiment, the version of software resident in a device's memory and the location and size of unused free memory is determined by initializing the device's memory with one or more pre-determined binary patterns. In one embodiment, buffer spaces are inserted between software modules in a device's firmware to allow for expansion of a particular software module by way of a future update. In one embodiment, one or more software modules (or objects) are inserted into pre-configured code expansion slots in device's firmware. The memory space that remains after all objects are inserted into a slot provides buffer space to allow for future updates to these objects.

[17] These and other advantages, aspects, and novel features of the present invention, as well as details of illustrated embodiments, thereof, will be more fully understood from the following description and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[18] Figure 1 is a block diagram of a memory initialization and software update system for an electronic device in accordance with an embodiment of the invention.

[19] Figure 2 is an operational flow diagram illustrating a method of initializing and updating software loaded in a device memory of an electronic device in accordance with an embodiment of the invention.

[20] Figure 3A is a diagram illustrating a memory image comprising a number of software modules and un-initialized free memory space configured for loading into an electronic device in accordance with an embodiment of the invention.

[21] Figure 3B is a diagram illustrating a device memory initialized with a pre-determined pattern in accordance with an embodiment of the invention.

[22] Figure 3C is a diagram illustrating a memory image comprising a number of software modules and initialized free memory space configured for loading into an electronic device in accordance with an embodiment of the invention.

[23] Figure 4A is a relational block diagram illustrating an arrangement of software modules and remaining free memory space in a device memory in accordance with an embodiment of the invention.

[24] Figure 4B is a relational block diagram illustrating an arrangement of software modules, a block of random free memory space between two software modules, and remaining free memory space in a device memory in accordance with an embodiment of the invention.

[25] Figure 5A is a relational block diagram illustrating a number of uniformly sized free memory spaces interspersed between a number of software modules in accordance with an embodiment of the invention.

[26] Figure 5B is a relational block diagram illustrating a number of software modules and corresponding free memory spaces in which the size of a free memory space is proportional to the size of its corresponding software module in accordance with an embodiment of the invention.

[27] Figure 6A illustrates an exemplary input that is processed by a layout processor to yield a scatter load file output in accordance with an embodiment of the invention.

[28] Figure 6B illustrates an exemplary scatter load output from a layout processor in accordance with an embodiment of the invention.

[29] Figure 7A illustrates the transfer of software from ROM to RAM in firmware without free buffer spaces.

[30] Figure 7B illustrates the transfer of software from ROM to RAM in firmware with free buffer spaces in accordance with an embodiment of the invention.

[31] Figure 8 illustrates groupings of object files based on change factors and their relationship to memory addresses in a device memory in accordance with an embodiment of the invention in accordance with an embodiment of the invention.

[32] Figure 9 provides an illustration of an exemplary scatter load file in accordance with an embodiment of the invention.

[33] Figure 10 is a block diagram illustrating objects and unused buffer spaces in firmware loaded on a device memory in accordance with an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[34] Figure 1 is a block diagram of a memory initialization and software update system in accordance with an embodiment of the invention. The memory initialization and software update system 105 comprises a generator 107 that generates a software update package for an electronic device 111 with device memory 110 by way of a memory layout specification, a software repository 113, an update agent 109 residing within the electronic device 111 that can apply the update package onto the electronic device 111 to upgrade any existing software resident in the device memory 110 of the electronic device 111, a layout preprocessor 115 that facilitates the introduction of unused memory space for software packages, a user interface 120 to allow input of parameters related to the memory layout specification, and a binary image creator 124 that generates binary image packages for initializing of the device memory 110 of the electronic device 111. The software repository 113 may store software packages generated by the generator 107, as well as binary image packages generated by the binary image creator 124. The software packages generated by the generator 107 comprise software required by the device memory 110 of the electronic device 111 to function properly. The software packages may comprise a number of software modules. Typically, the software (or firmware) is executed upon power up of the electronic device 111 in preparation for normal operation by the user. An external computer system 128 may be used as a source to provide a memory layout specification to the layout preprocessor 115. The terms software package, update package, and software update package used herein includes instructional code that may be processed by the update agent 109 in the creation of updated software in the electronic device 111. Herein, the concept

of initializing a device memory 110 by a binary pattern occurs by way of creating a binary data image that is subsequently loaded onto the device memory 110 of the electronic device 111.

[35] The electronic device 111 comprises any electronic device such as an exemplary mobile handset, personal digital assistant (PDA), computer, pager, interactive digital appliance, or the like. Within the electronic device 111 resides a device memory 110 capable of storing and running firmware or software for properly booting up and subsequently operating the electronic device 111. Exemplary device memory 110 includes flash memory, random operating memory (ROM), electrically programmable ROM, or the like.

[36] The software repository 113 may comprise one or more computer storage devices such as a hard drive, optical drive, CD-ROM drive, DVD drive, tape drive, or other like device. As illustrated in Figure 1, the software repository 113 interfaces with the binary image creator 124 and the generator 107.

[37] The update agent 109 comprises hardware and / or software configured to provide processing of a software package provided to it by the generator 107. The update agent 109 may reside within the electronic device 111.

[38] The external computer system 128 comprises one or more computers in a network providing one or more storage devices with storable media. The external computer system 128 interfaces with the binary image creator 124 or layout preprocessor 115.

[39] The user interface comprises a keyboard or other input device, as well as a monitor or visual communications device capable of permitting a user to input commands and data into the software update system 105.

[40] The binary image creator 124 configures a binary image by way of a memory layout specification for a given device memory 110. The binary image creator 124 may comprise a combination of hardware and software capable of generating binary image patterns. It is contemplated the memory layout specification may be obtained from the device memory 110 or from a storage device within an external computer system 128 or by manual input by way of a user interface 120 connected to the layout preprocessor 115. The memory layout specification provides information regarding the characteristics of the device memory 110 in the electronic device 111. In addition, the size of the unused memory space to be deployed and the location or position of these spaces comprise the memory layout specification.

[41] The layout preprocessor 115 comprises hardware and / or software capable of utilizing a memory layout specification. It generates a layout configuration file often called a scatter load file (to be discussed later) that maps the memory locations of software modules loaded into the device memory 110. The configuration file is then provided to the generator 107 in which a suitable software update package is generated for the electronic device 111.

[42] The generator 107 comprises hardware and software capable of generating a software package for direct incorporation into a device memory image or may store the software package in the software repository 113 for subsequent retrieval and use by one or a plurality of software update systems 105. It is contemplated the software package comprises either a new software package that is loaded in a new electronic device 111 at the time of manufacture or a software update package that is installed as a revision to an existing software package in an electronic device 111. Incorporation of the software package into the device memory 110 is by way of an update agent 109 resident within the electronic device 111. The generator 107 typically creates a

software package by comparing an existing software image resident in the device memory 110 of the electronic device 111 to a newer version of the same software that may be stored in the software repository 113 or the external computer system 128. The generator 107 computes differences in the software images and creates an appropriate software package. The update agent 109 in the device 111 is capable of applying the update package onto the electronic device 111 by processing and executing the instructions provided with the software package.

[43] The generator 107 determines that an electronic device 111 requires a software update by way of a binary image previously stored onto the device memory 110 of the electronic device 111 by way of the binary image creator 124. The binary image package may be generated by the binary image creator 124 for use by the generator 107 in the incorporation of new or updated software into the device 111. The binary image package generated by the binary image creator 124 and the software package generated by the generator 107 may be installed into a plurality of electronic devices that are similar or identical in design and function to the electronic device 111. The binary image package may be directly transmitted to the electronic device 111 by the binary image creator 124 for installation into the device memory 110. It is contemplated the update agent 109 may facilitate installation of the binary image. Different binary image packages comprising distinct binary patterns may be created. One or more of these patterns may be installed in a plurality of one or more types of devices as necessary. When interfaced with the generator 107, the different versions of installed binary images in the plurality of electronic devices may be easily analyzed or compared by the generator 107 in order to determine differences in software version. Further, the implementation of these unique binary images or patterns may allow the generator 107 to differentiate unused memory space from software

resident in the device memory 111 and calculate any available free or unused memory space. It is contemplated the generator 107 may interface with an electronic device 111 by way of wireless or wireline data communications.

[44] By populating the unused portion of memory with binary patterns characterized by a known and consistent value, a version of software that exists in a particular electronic device 111 may be identified. Further, a comparison of different versions of software resident in a device memory 110 should yield a specific difference image. For example, a generator 107 that is configured to process a difference image may subtract a version n image from a version n+m image to generate a resulting difference image, necessitating that a specific software update package be generated and incorporated into all electronic devices using a version n image.

[45] In general, the binary image creator 124 creates a binary memory image for initialization of the device memory 110. A finalized binary image may be loaded into the device memory 110 by the binary image creator 124 before any software package is loaded onto the device memory 110 by the generator 107. In other instances, software modules may be incorporated into the binary memory image prior to loading. In one embodiment, the binary image creator 124 is responsible for assembling a finalized memory image that is subsequently programmed or flashed into the device memory 110 of the electronic device 111, such as during manufacturing. Furthermore, it is contemplated that predetermined binary images may be stored within the software repository 113 for future use. Predetermined binary images may be comprised of a pattern of binary or hexadecimal characters that create a unique image. For example, the device memory 110 may be initialized with the value 0xFFFF throughout its entire memory. Other values or combination of values may be employed in other embodiments. In one embodiment,

the memory of the device memory 110 is segmented into multiple memory sections where each memory section is selectively initialized with a different predetermined memory pattern.

[46] It is contemplated that various software packages representing different versions of software update packages may be saved in the software repository 113 by the generator 107 or loaded directly onto device memory 110 of one or more electronic devices 111. In one embodiment it is contemplated a software package may be incorporated into a binary image and then saved into the software repository 113 for future use by the generator 107. In one embodiment of the present invention, the values of the binary image pattern are consistent across electronic devices 111 with the same software or firmware version. The update agent 109 may serve to facilitate the incorporation of the binary image or software modules into the device memory 110.

[47] In general, the binary image creator 124 may be used to initialize device memory 110 in different types of devices so that subsequent processing by the generator 107 may be accomplished. Such processing includes determining different versions of the same software or determining unused memory locations in a device memory 110.

[48] Figure 2 is a process flow diagram illustrating an exemplary process of memory image initialization of device memory and subsequent loading of software packages, in accordance with an embodiment of the invention. At step 209, a binary image creator 124, such as that illustrated in Figure 1, selectively retrieves a memory layout specification from an external computer system 128, from a device memory 110, or from user input through a user interface 120. Information such as number of memory, type of memory, and size of memory is provided by the memory layout specification for use by the binary image creator 124. Next, at step 211, an

appropriate binary image pattern configured for use with the memory layout specification is selected and retrieved from a binary image repository 113. Then, at a next block 213, the binary image creator 124 may create one or more images for initializing one or more device memories. The binary image creator 124 accomplishes this by writing or loading the one or more images it creates into one or more electronic devices 111. In other instances, the binary image creator 124 may configure a binary image comprising a pattern suitable for loading into a device memory for eventual or future use. This binary image may be stored in the software repository 113. Subsequently, at step 215, a software repository 107 provides an image of one or more software packages or software module(s) to the electronic device 111 for incorporation into the device memory 110. Software associated with functionality of the update agent 109 may be incorporated into memory by way of modification of the binary image. The resulting image in memory is such that unused or free memory can be identified by means of the initialized pattern(s). In addition, a particular binary pattern may be used in the initialization process to correspond with a particular software package version, type of software, or types of software modules loaded into the device memory 110. In this fashion, a particular software version resident in device memory 110 may be identified. Later, at steps 217 and 219, the initialized binary images with written software module(s) are saved in a software repository 113 or loaded into one or more device memories 110 of electronic devices 111.

[49] Figure 3A is a block diagram illustrating exemplary software modules and unused free memory space resident in an un-initialized device memory 307 of an electronic device, in accordance with an embodiment of the invention. Exemplary software modules A 309, B 313 and C 315 are distributed in the un-initialized memory 307 with some amount of free memory

available 311, 317, although the available free memory 311, 317 may be indistinguishable from memory occupied by the software modules A 309, B 313, and C 315, because the unused free memory 311, 317 is characterized by a random bit pattern. In addition, a comparison of images resident in un-initialized memories of two similar electronic devices by the generator 107 is likely to produce inconsistent difference images. As illustrated in Figure 3A, unused free memory 317 is typically located at the end of the un-initialized memory 307 and an occasional unused free memory block 311 may occur at random. Software modules A 309, B 313, and C 315 may have adjacent free memory blocks introduced intentionally or by random chance.

[50] Figure 3B is ablock diagram illustrating an exemplary memory image of an initialized device memory 325, in accordance with an embodiment of the invention. Here, the entire memory is initialized with a unique pre-determined pattern. After initialization, the unused memory 327 spans the entire device memory. Because the pattern is known and deterministic, any unused or free memory space within a device memory is easily detected after a software package is loaded into the initialized device memory 325. In addition, the pre-determined pattern provides a recognizable image that may be used as a tag or identifier to identify a particular software version. When comparing a device memory of an earlier software version with a current release version, the exemplary generator 107 shown in Figure 1 may generate a unique difference image and subsequently correlate this with a corresponding software update package. In one embodiment of initializing a memory, a specific pattern may be employed for an entire device memory. In another embodiment, the device memory is segmented into multiple segments, and different patterns are stored into the different segments of the memory during

initialization. In the latter method, it is contemplated the different patterns provide additional means to identify an image.

[51] Figure 3C is a block diagram of an exemplary embodiment in accordance with the present invention, illustrating software modules A 359, B 363, and C 365 written into initialized device memory 357. The software modules A 359, B 363 and C 365 are written onto memory that is initialized with a specific pattern making it possible to distinguish the available free memory 361 367 from memory currently occupied by software modules 359 363 365. In addition, the pre-determined pattern provides an image that may be used as a tag or identifier to identify a particular software version.

[52] The electronic device 111 may determine a priority of use based upon the specific bit pattern employed in the unused free space. The electronic device 111 may consume portions of unused memory accordingly. For example, one or more portions of unused memory may be coded with a specific binary pattern indicating a priority or preference of use. If the electronic device 111 eliminates any software module, or if any software module relinquishes currently used memory, the released memory space can be selectively re-initialized with one or more specific patterns designating its preference for future use.

[53] In one embodiment, the device memory 110 comprises a FLASH type of memory initialized entirely to a specific pattern comprising hexadecimal values of 0xFFFF. In another embodiment, the device memory 110 comprises four FLASH segments (FLASH1, FLASH2, FLASH3, FLASH4) specified to be 0x20000 in size, plus a RAM segment of size 0x40000. In one embodiment, the FLASH segments and the RAM segment are all initialized to the same

pattern. In one embodiment, the FLASH and RAM segments are initialized using one or more patterns.

[54] The layout preprocessor 115 determines available free space in the device memory 110 by way of a memory layout specification provided by the external computer system 128 or one or more electronic devices 111. The preprocessor 115 distributes available free space among individual software modules so as to provide effective and efficient updating of one or more software modules resident in the device memory 110. If the device 111 has extra free space that is currently not being used, the extra free space can be distributed among the available software modules and the software modules can be spread out in memory so as to provide free space segments, called reserved buffer spaces, juxtaposed to the software modules, providing a mechanism to constrain any update related changes to an individual software module and any dedicated free space around it.

[55] In one embodiment of the present invention, one or more memory layout specifications are contained in the layout preprocessor 115 and may be selectively employed by the layout preprocessor 115 to distribute available free space among software modules in a device memory 110 by creating one or more reserved buffer spaces adjacent to each of the software modules. In another embodiment, a memory layout specification is retrieved by the layout preprocessor 115 from an external system 128, such as was described in Figure 1.

[56] In one embodiment, the memory layout specification specifies the layout of one or more sections of memory on the device, wherein the memory is comprised of one or more sections of FLASH memory and RAM memory. The origin and length of each section of FLASH or RAM is also specified along with assignments of one or more software modules to individual memory

sections. The mapping of specific groups of code within a software module loaded onto the various sections of the memory (FLASH RAM) of the device are also specified. The layout preprocessor 115 incorporates the information contained in the memory layout specification in creating the reserved buffer spaces.

[57] In one embodiment, the layout preprocessor 115 reorganizes and locates software modules on the electronic device 111 by computing the available free space and distributing this free space equally among the various software modules as reserved buffer spaces. In general, the layout preprocessor 115 is used as a pre-processing tool in configuring device memory images for different types of electronic devices 111 so that subsequent processing of one or more software modules by the generator 107 is easily performed with minimal change to unmodified software in the device memory 110. The layout preprocessor 115 may insert free spaces into one or more selected software modules in memory as a means to effectively localize changes to software that require additional free space.

[58] Figure 4A is a block diagram of an exemplary arrangement of software modules in a device memory 405 wherein the software modules A, B, C and D 407, 409, 411, 413 are positioned adjacent to each other without any free memory space separating the modules, according to an embodiment of the invention. Beyond the last module 413 is an unused or free memory segment 415 representing the remainder of unused memory. The software modules A 407, B 409, C 411, and D 413 may or may not be related to each other. Being adjacent to each other without any free memory between may cause a reorganization of links to addresses and references in other modules in order to modify or upgrade any one module . For example, if a new version of software module A 407 is loaded to replace the existing software module A 407,

then replacing the existing version with the new version may require more or less memory space than what is currently utilized. Should additional memory space be required, a relocation or shifting of some or all modules B 409, C 411, and D 413 may occur. Relocation of software modules often necessitates changing the addresses of software jumps and references to addresses. Thus, an avalanche or overflow effect may occur in which an update to software module A 407 results in changes to successive modules B 409, C 411, and D 413. Updates or revision to the last software module 413 may not produce an avalanche or overflow effect since it is adjacent to the free memory space 415 at the end of memory.

[59] Figure 4B is a block diagram of another exemplary layout of software modules in a memory displaying software modules that are adjacent to each other without free memory between them, according to an embodiment of the invention. As shown, a free memory segment 435 is characteristically located at the end of the memory block 425, and an occasional free memory block 437 may occur at random. It is contemplated the occasional free memory block 437 provides a small free memory space. Specifically, software modules A 427, B 429 and C 431 are adjacent to each other and modules A 427, B 429 do not have any adjacent free memory blocks while software module C 431 has a free memory block 437 adjacent to it, introduced not intentionally, but by random chance.

[60] Such a memory layout may cause an avalanche or overflow effect described above for Figure 4A, in which changes to any software module due to version upgrades are likely to cause relocation of one or more software modules downstream in the device memory 110. Relocation and shifting of memory may occur despite the occurrence of random free memory blocks 437

because they might not be large enough to accommodate the size requirements of software upgrades due to version changes, patches, or bug fixes.

[61] Figure 5A is a block diagram of an exemplary layout of software modules created by the layout processor 115 for a device memory 110 wherein uniform sized reserved buffer spaces are placed between the installed software modules A 507, B 509, C 511, and D 513, in accordance with an embodiment of the invention. By introducing reserved buffer spaces 517, 519, 521 and 533 adjacent to software modules A 507, B 509, C 511, and D 513, respectively, the layout processor 115 creates a layout for the software modules that accommodates changes to individual software modules without shifting other installed software modules. In fact, the size of each of the reserved buffer spaces may be adjusted so as to make relocation or shifting of other software modules unnecessary.

[62] In one embodiment, the reserved buffer spaces 517, 519, 521 and 533 are configured to be the same size. In another embodiment, the reserved buffer spaces 517, 519, 521 and 533 vary based on the characteristics of the individual software modules A 507, B 509, C 511 and D 513.

[63] Figure 5B is a block diagram of an exemplary layout of software modules created by the generator 107 for a device memory 110 wherein non-uniform sized reserved buffer spaces 517, 519, 521 and 533 are positioned between the installed software modules so as to minimize the shifting effects associated with upgrading one or more software modules, according to an embodiment of the invention. Here, the size of each reserved buffer space is proportional to the size of the software module(s) it is associated with. For example, the size of reserved buffer space 517 adjacent to the software module A 507 is selected based upon and in proportion to the size of the corresponding software module A 507. Thus, reserved buffer space 519 is smaller

than reserved buffer space 517 because the software module B 509 is smaller than the software module A 507.

[64] The methods described in Figures 5A and 5B illustrate an embodiment where reserved buffer spaces are placed between all software modules. Hence, each software module has its own dedicated free memory space.

[65] Figure 6A illustrates an exemplary input instructional software code that is processed by the layout preprocessor 115 to generate a memory mapping or scatter load file (as will be discussed later) that incorporates reserved buffer spaces between software modules in the device, in accordance with an embodiment of the invention. It depicts a memory that comprises a FLASH segment of size 0x70000 and a RAM segment of size 0x40000. The software code incorporates information provided by a memory layout specification.

[66] Figure 6B is an exemplary output of a scatter load file that is created by the layout preprocessor 115 that incorporates reserved buffer spaces between software modules in a device, according to an embodiment of the invention. It depicts a memory that comprises four FLASH segments, with a total size 0x70000, with one FLASH segment (FLASH1) specified to be 0x1000 in length and three other FLASH segments (FLASH1, FLASH1, FLASH1) specified to be 0x20000 in size, in addition to a RAM segment of size 0x40000. For each of the FLASH segments, a mapping of individual software modules is also depicted by way of a data structure termed SECTIONS.

[67] When a new software module update is generated by the generator 107 (as illustrated in Figure 1) for an electronic device 111, additional memory space may be required beyond the

space occupied by the existing software module. One method to provide reserved buffer spaces within a device memory of an electronic device is by allocating slots within memory in which one or more software modules or object files may be loaded. These slots may be filled with other object files as new or updated code is incorporated to the electronic device. It is contemplated the slot sizes would be designed to accommodate any anticipated expansion due to expected software updates. Using a design technique described below, the expansion slots are not completely filled by successive updates. Hence, successive slots in memory are not be affected by overflow from a preceding slot obviating any shifting or relocation of software code.

[68] An embodiment of the layout preprocessor 115, termed a code expansion slot tool (CEST), may be used to implement the slot allocation technique described in the preceding paragraph. In this embodiment, a scatter load file is employed to configure memory layout information of software modules stored within a device memory 110. In this embodiment, the scatter load files provide configuration information concerning the assignment of software modules (or objects) to slots and the amount of memory space assigned per slot. Any reserved buffer spaces are determined for each slot based on remaining space after all objects have been assigned to a particular slot. It is contemplated the CEST comprises a software application with graphical user interface that operates on the scatter load file and is executed at the layout preprocessor 115 (illustrated in Figure 1). The scatter load file is ultimately used by the generator 107 in creating a software package for incorporation into the device memory 110.

[69] The user of the CEST may specify the number of slots a memory image should be divided into by way of the user interface 120. The user of the CEST may organize the memory slots within the image according to a likelihood of change of the objects within a slot. In one

embodiment, the objects that are likely to change may be placed within a slot closest to the end of memory. In one embodiment, the objects that are likely to change may be placed within a slot of large memory size. These scatter load files may be stored on a per project basis in the external computer system 128 or in the layout preprocessor 115. It is contemplated the CEST may be used as a tool to write or formulate an appropriate scatter load file. If the CEST determines that the user does not have enough memory (i.e., ROM) to partition the entire software into the number of slots required, the CEST may be configured to provide an error message and a diagnostic report.

[70] Figure 7A is a perspective block diagram that depicts exemplary ROM 707 and RAM 709 footprints in a typical mobile handset without the use of the methods described in connection with this invention. During operation, software code from ROM 707 (such as in a FLASH memory) is copied to a designated address location in RAM 709 for execution. In one embodiment, the addresses and references of the objects in ROM correspond to their actual designated locations in RAM during execution. As illustrated, the application software 701 in ROM is copied to RAM as identical application software 702.

[71] In contrast, Figure 7B is a perspective block diagram that depicts exemplary ROM 711 and RAM 713 footprints in a mobile handset wherein code expansion slots are employed to allow for growth in software code due to updates, in accordance with an embodiment of the invention. The application software 715, 716, 717 in ROM is identically transferred to RAM as application software 725, 726, 727. As a consequence, the mobile handset operates identically to that described in Figure 7A.

[72] Figure 8 is a perspective block diagram that depicts an exemplary grouping of software objects that is grouped based on their probability of change, according to an embodiment of the invention. The probability of change is termed a change factor. As illustrated, objects having like probability of change are grouped into the same change factor. In this exemplary diagram, the change factor varies from a value of 0 to a value of 3. As illustrated, the objects having the greatest likelihood of change are placed towards the end of memory (at highest address locations) and are assigned a change factor of 3. As shown by the directional arrow 820, the change factors having a higher probability of change are stored in higher address memory locations in ROM. In this exemplary diagram, ROM is divided into four slots labeled APP_ROM1 807, APP_ROM2 808, APP_ROM3 809, and APP_ROM4 810 to accommodate object files with varying change characteristics. Object files Bootapp.o, audfmt.lib, saf.lib, bfcomdrv_null.o, NpComDrv_Common.o, and dsmgr.o are associated with a zero change factor and are characterized by a low probability of change. As a consequence, they are stored in a low address memory range such as APP_ROM1. On the other hand, object files such as dadownloadagent.o, dachunk.o, daheap.o, daprotocol.o, dsctl.o, and bfbearerlib_null.o are stored in APP_ROM4 810 because these files are considered to have a high probability of change.

[73] Typically, a firmware image may comprise thousands of objects to allow for a fully functional device. When a firmware comprises many thousands of object files, it is contemplated the update methods employed by the embodiments described is of greater significance because of the avalanche or overflow effect described. When such methods are not employed, an update may necessitate using an update package of increased size.

[74] As the amount of available space in memory is limited, it is useful to allocate memory space as effectively as possible. Updates are often necessitated by major bug fixes, performance requirement changes, or new functionalities. Often a user's practical experience is employed to refine the amount of reserved buffer spaces allocated per block of device memory to accommodate for such updates. If experience dictates that a particular object is not going to change during a device's lifetime, it may be desirable to assign it to a slot containing little unused space unless extra space is plentiful. In one embodiment, a weighting factor may be used to multiply an average buffer slot size in the calculation of the amount of reserved buffer space or code expansion space required. It is contemplated that by experimentation and experience, a user may determine the average buffer slot size required.

[75] The selection criteria for grouping objects are also addressed. For example, all objects that were predicted not to change may be grouped together into a common slot at the beginning of memory. In addition, if two or more objects share similar functionality, they may be grouped together. In this instance, it is contemplated the benefit is easier organization and implementation of a particular design.

[76] In one embodiment, a firmware image intended for release is built by compiling code and organizing the compiled objects into an image. During the build process, a linker generates what may be called a map file. A map file contains the results of the link including the final sizes of objects and their placement in the firmware image. From the map file, the names and sizes of all objects are defined and this information may be subsequently assigned into a device memory's slots. In some cases, objects can be grouped and in other cases single objects could be placed in dedicated slots. As an example, an object A is found in the map file to occupy 24KB. Because

the probability of change for this object is considered high, it is estimated the object might grow to 32KB in the future. Based on this value, a slot size of 32KB may be specified in its corresponding scatter load file. With the slot sizes preliminarily defined by a map file, a scatter load file can be used to re-specify each slot or software module size and its associated object files. In one embodiment, a linker is used to place compiled source code into software (firmware). The linker arranges all the compiled objects into a firmware image of which the order of objects in firmware are specified by the scatter load file. This image is then copied into a flash memory of an electronic device. To implement an appropriate layout of memory, a scatter load file may be formatted as follows:

Scatter Load File (SLF) Format

```
section-name starting-address size
{
        object file-list
}
```

and has the following definitions:

section-name — An arbitrary label that describes a section, which can be considered a slot.

starting address — Address that specifies where in the final firmware memory the section is placed.

size— The size given to that section.

object file-list — The list of objects to be placed in the section or slot

[77] Figures 9 and 10 illustrate a set of instructions implementing an exemplary scatter load file 905 and its corresponding mapping of object files 1010 1012 1014 1016 and buffer spaces

1020, 1021, 1022 onto a firmware 1007, in accordance with embodiments of the invention. The scatter load file 905 specifies that an object A 910 is placed in section A 914, objects B 920 and C 922 are placed in section B 924, and an object D 930 is placed in section C 934. Section A 914 may accommodate up to 0x1000 bytes, while section B 924 up to 0x3000 bytes, and section C 934 up to 0x2000 bytes. In the case of section B 924, there are two objects of which the sizes of both objects B 920 and C 922 cannot exceed 0x3000 bytes. If the objects in any section exceed the section size, a linker error may be generated during the build of the firmware.

[78] In one embodiment, a code expansion slot tool (CEST) of the layout preprocessor 115, shown in Figure 1, may be configured to provide the following:

- A per project configuration file that may comprise information such as:
  - Number of memory blocks a device memory is divided into
  - Change factor values for each memory block
  - Scatter load file information such as filename and location.
  - Memory size
  - Starting and ending addresses in each block of device memory
  - Amount and location of reserved buffer space or code expansion slot for each block of device memory
  - Map filename and location
  - The names of the object files and what blocks they belong to
- Automated or manual creation of a scatter load file incorporating code expansion slots
- A configuration file for the general tool use that will store the names and locations of each project

- Ability to read Map files and determine the following:
    - Name of each load and execution block
    - Size of each load and execution block
    - Name and size of each object file
- A graphical user interface provided by the CEST
- Provide the user with the ability to input starting and ending information of blocks within device memory
- Provide the user the ability to associate one or more groups of object files with a change factor. It is contemplated an exemplary numeric scale for this factor will be as follows:

    0. No changes
    1. Infrequent Changes
    2. Moderate Changes
    3. Frequent Changes
- Ability to save a user's personal settings into the project configuration file
- Allocation algorithms for configuring memory blocks or slots within a device memory:
    - Allocate object files to memory blocks or slots according to its associated change factor such that memory blocks (or slots) at the highest addresses in memory have the greatest available free space
    - Allocate free space evenly among the device memory blocks
    - Allocate no space for the first two address blocks or slots in device memory, but evenly split the unused free memory space among the remaining memory blocks

  - Allocate space based on weighting factors correlated to probability of change and an average reserved buffer size
  - It is contemplated other combination of uniform and non-uniform space allocation methodologies may exist and may be implemented
- Provide an errata log which comprises the following:
  - Date in which error occurs
  - Project name associated with error
  - Error message
  - Diagnostic information
- Prompt the user concerning the allocation of single large blocks of device memory for a large software module such as an exemplary main application software module. It is contemplated the CEST may divide this block into a number of sub-blocks. The CEST may alert the user if a block is of a certain maximum size, in order to ensure that block sizes are not too large.
- Upon segmentation of large blocks, files may be assigned to the newly segmented memory blocks based on change factor.
- Ability of a user to manually map object files or other files with specific ROM blocks.

[79] The code expansion slot method utilizing a scatter load file as embodied in the CEST may provide minimization of update package sizes for many version releases of a firmware image. Just as an initial release requires code change factor estimates and heuristics that are experimented with until a desired slot organization and characteristics are achieved, during any

revision work, the slots should not be adjusted until all the changes are implemented and desired characteristics are determined and achieved.

[80] Organizing object code within firmware generally involves evaluating each software object and identifying the objects that are most likely to require modifications or enhancements. This determination may be based on a manufacturer's development cycle, product road map, or other distinguishable metrics. The objects with a high probability of significant modification would then be placed at the top of the firmware image or highest memory address in firmware. If modifications are required after the device's commercial release that result in significant changes to these objects, displacement, relocation, or shifting of objects into other memory blocks or slots would most likely occur. Further, the addition of large updates may overwhelm the reserved memory in one or more slots. However, since these objects were placed at the top of the image, the avalanche effect is limited to the top area of flash, resulting in a smaller update package (used for updates) than would have been generated (by a generator 107) if these objects were located at the bottom of the image.

[81] In summary, aspects of the present invention provide a number of embodiments of a system and method to effectively initialize and update firmware in an electronic device. In one embodiment, the version of software resident in a device's memory and the location and size of unused free memory are determined by initializing the device's memory with one or more pre-determined binary patterns. In one embodiment, buffer spaces are inserted between software modules in a device's firmware to allow for expansion of a particular software module by way of a future update. In one embodiment, one or more software modules (or objects) are inserted into pre-configured code expansion slots in device's firmware. The memory space that remains after

all objects are inserted into a slot provides buffer space to allow for future updates to these objects.

[82] While the invention has been described with reference to certain embodiments, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted without departing from the scope of the invention. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the invention without departing from its scope. Therefore, it is intended that the invention not be limited to the particular embodiment disclosed, but that the invention will include all embodiments falling within the scope of the appended claims.

CLAIMS

WHAT IS CLAIMED IS:

1. A system for initializing a memory and updating software in an electronic device comprising:

a generator for generating a software package for incorporation into a device memory of said electronic device;

a binary image creator for generating a binary pattern for writing into said device memory of said electronic device;

a layout preprocessor for implementing a memory layout specification into software code, mapping configuration parameters associated with said layout specification, and incorporating reserved buffer spacing for use by said generator;

a software repository for storing said software package or said binary pattern; and

a user interface for inputting said layout specification directly into said layout preprocessor.

2. The system of Claim 1 further comprising an external computer system for providing a layout specification to said layout preprocessor.

3. The system of Claim 1 wherein the software repository comprises one or more computer storage devices.

4. The system of Claim 1 wherein said layout processor is configured to insert a buffer space between consecutive said software modules for subsequent processing by said generator.

5. The system of Claim 1 wherein said layout processor is configured to assign one or more software modules to one or more predefined slots of memory for subsequent processing by said generator.

6. The system of Claim 5 wherein said software modules comprise software objects.

7. The system of Claim 4 wherein said corresponding buffer spaces are of equal size.

8. The system of Claim 4 wherein said corresponding buffer space comprises a size proportional to its associated software module.

9. The system of Claim 5 wherein said layout processor assigns one or more software modules to one or more predefined slots of memory based on a probability of change of said one or more software modules.

10. The system of Claim 9 wherein said probability of change is specified by a numeric value.

11. The system of Claim 5 wherein sizes of said predefined slots is based on a probability of change of said one or more software modules.

12. The system of Claim 5 wherein sizes of said predefined slots is based on a weighting factor correlated to probability of change and an average reserved buffer size.

13. The system of Claim 5 wherein the first two slots of said one or more predefined slots of memory provide no buffer space and the remaining slots provide buffer spaces of equivalent sizes.

14. The system of Claim 1 wherein said generator identifies versions of software modules residing in said device memory and generates a suitable software update package for processing by said update agent into said device memory in said electronic device.

15. The system of Claim 1 wherein said mapping of software modules by said preprocessor minimizes the size of said software packages.

16. A method of updating software in a device memory of an electronic device comprising:

obtaining a memory layout specification;

processing said layout specification;

generating a software update package for loading into said electronic device; and

processing said update package within said electronic device to form a new image into said device memory.

17. The method of Claim 16 wherein said generating a software update package comprises:

assessing an image provided by a device memory of an electronic device;

comparing said image to one or more images stored in a software repository;

subtracting said two images to produce a difference image; and

generating an image for said software update package based on said difference image.

18. The method of Claim 116 wherein said obtaining layout specification is performed by way of input of data from a user through a user interface.

19. The method of Claim 16 wherein said obtaining layout specification is performed by accessing said layout specification data from an external computer system.

20. The method of Claim 16 wherein said obtaining layout specification is performed by accessing said layout specification data from one or more electronic devices.

21. A method of identifying free space within a device memory of an electronic device comprising:

obtaining a memory layout specification;

generating one or more predetermined binary pattern(s) based on said memory layout specification; and

initializing said device memory with said binary pattern(s).

22. The method of Claim 21 wherein said initializing is performed by an update agent located within said electronic device.

23. The method of Claim 21 wherein said obtaining memory layout specification is performed by way of input of data from a user through a user interface.

24. The method of Claim 21 wherein said obtaining memory layout specification is performed by accessing said layout specification data from an external computer system.

25. The method of Claim 21 wherein said obtaining memory layout specification is performed by accessing said layout specification data from one or more electronic devices.

26. A method of differentiating versions of one or more software modules stored in a device memory of an electronic device comprising:

initializing a device memory with one or more predetermined binary patterns;

loading one or more predetermined software modules onto said device memory;

providing free space for one or more software modules in said device memory; and

storing a memory image of result of said initializing, loading, and providing, into a software repository.

27. A method of minimizing shifting or relocation of software addresses and references within a device memory of an electronic device comprising providing free space between successive software modules to accommodate for future expansion.

28. A method of minimizing shifting of software addresses and references within a device memory of an electronic device comprising:

determining an amount of memory space required by one or more software modules resident in said electronic device;

determining total size of memory space available in device memory from a memory layout specification;

calculating an available free space;

grouping of said one or more software modules into one or more slots based on one or more factors; and

assigning a fraction of said available free space to one or more slots based on one or more parameters.

29. The method of Claim 28 wherein said one or more factors is based on similar functionality of said one or more software modules.

30. The method of Claim 28 wherein said one or more parameters is based on probability of change of said one or more software modules.

31. The method of Claim 28 wherein said grouping of said software modules into one or more slots is based on probability of change of said one or more software modules.

32. The method of Claim 28 wherein said grouping and assigning is performed by implementing a software code incorporating said memory layout specification.

33. The method of Claim 32 wherein said software code comprises a scatter load file.

34. A system for creating a memory image for an electronic device comprising a layout preprocessor for introducing free buffer spaces between a plurality of software modules in said memory image.

35. The system of Claim 34 further comprising a binary image creator.

36. A system for reorganizing a plurality of software components in a binary memory image of an electronic device, the system comprising:

a layout preprocessor capable of processing the binary memory image;

a binary image creator capable of creating an output binary image of the electronic device based on code for the plurality of software components and a scatter load file;

a code expansion slot tool capable of incorporating code expansion buffers for the future expansion of selected ones of the plurality of software components; and

the system being capable of processing and reorganizing the binary memory image such that subsequent updates to any one of the plurality of software components will not require the shifting of some of the others of the plurality of software components.

37. The system according to claim 36 wherein the processing and reorganizing by the system of the binary memory image comprises creating a reorganized scatter load file, introducing buffer spaces between the selected ones of the plurality of software components as per the reorganized scatter load file, and generating an output binary image that corresponds to the reorganized scatter load file.

38. The system according to claim 36 wherein the layout preprocessor creates a reorganized scatter load file based on an initial memory map and information retrieved from processing the binary image and communicates the reorganized scatter load file to the binary image creator for the outputting of a binary image that corresponds to the reorganized scatter load file.

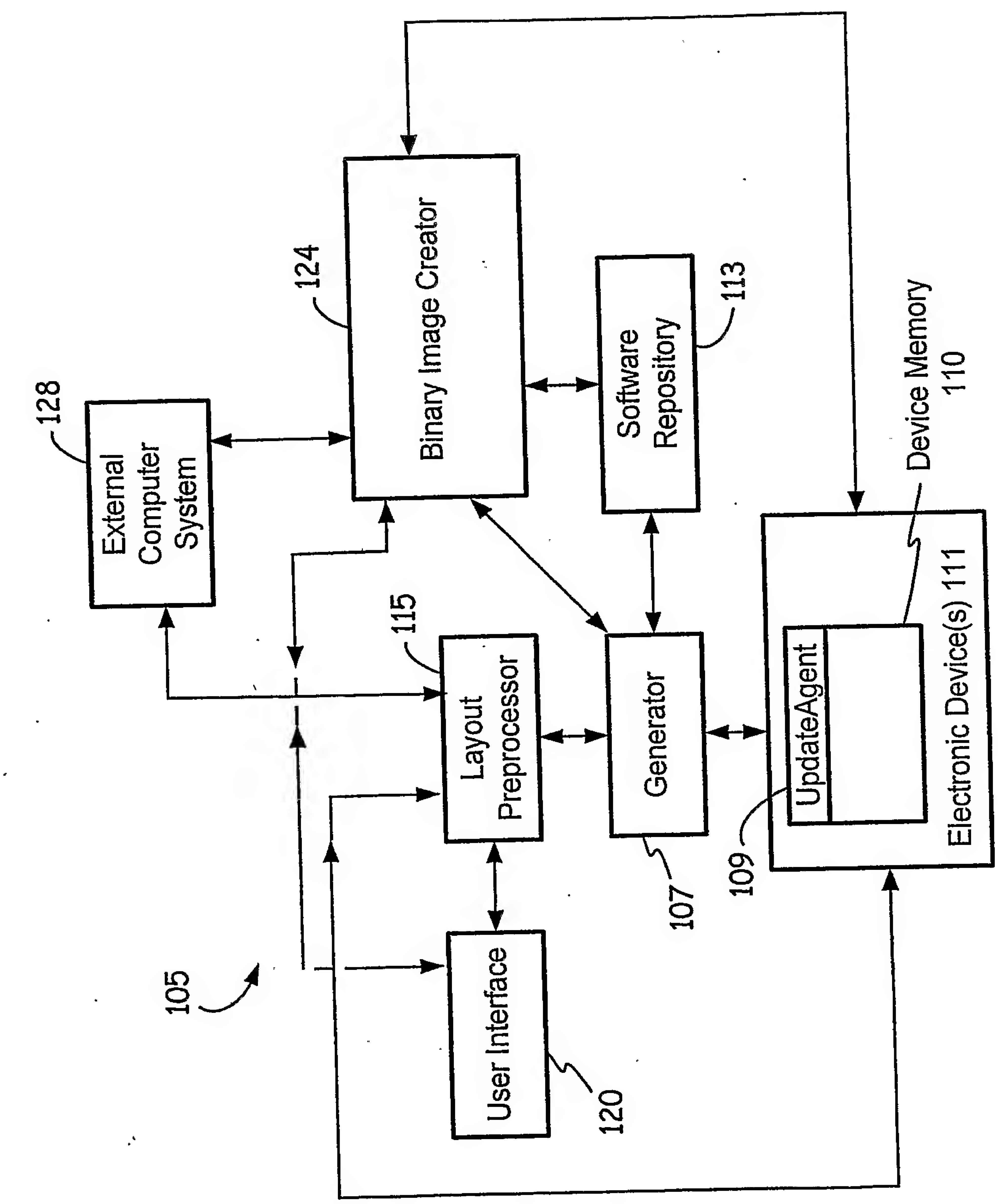

External Computer System
128
Binary Image Creator
124
Software Repository
113
Device Memory
110
Layout Preprocessor
115
Generator
107
UpdateAgent
109
Electronic Device(s) 111
User Interface
120
105

Fig. 1

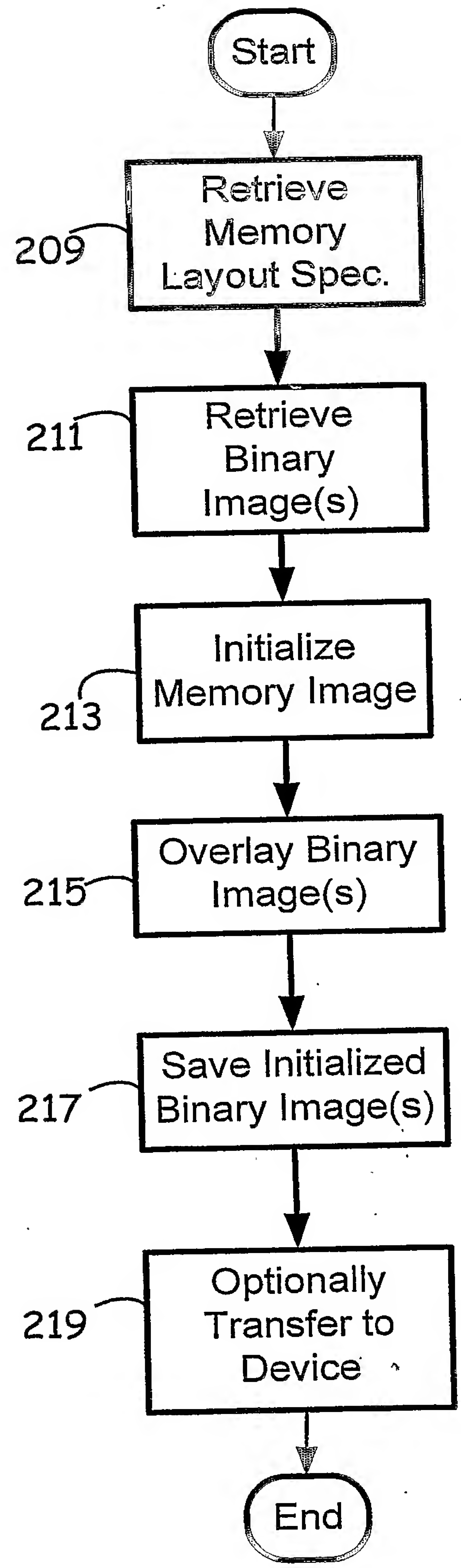

Start
Retrieve Memory Layout Spec.
209
Retrieve Binary Image(s)
211
Initialize Memory Image
213
Overlay Binary Image(s)
215
Save Initialized Binary Image(s)
217
Optionally Transfer to Device
219
End

Fig. 2

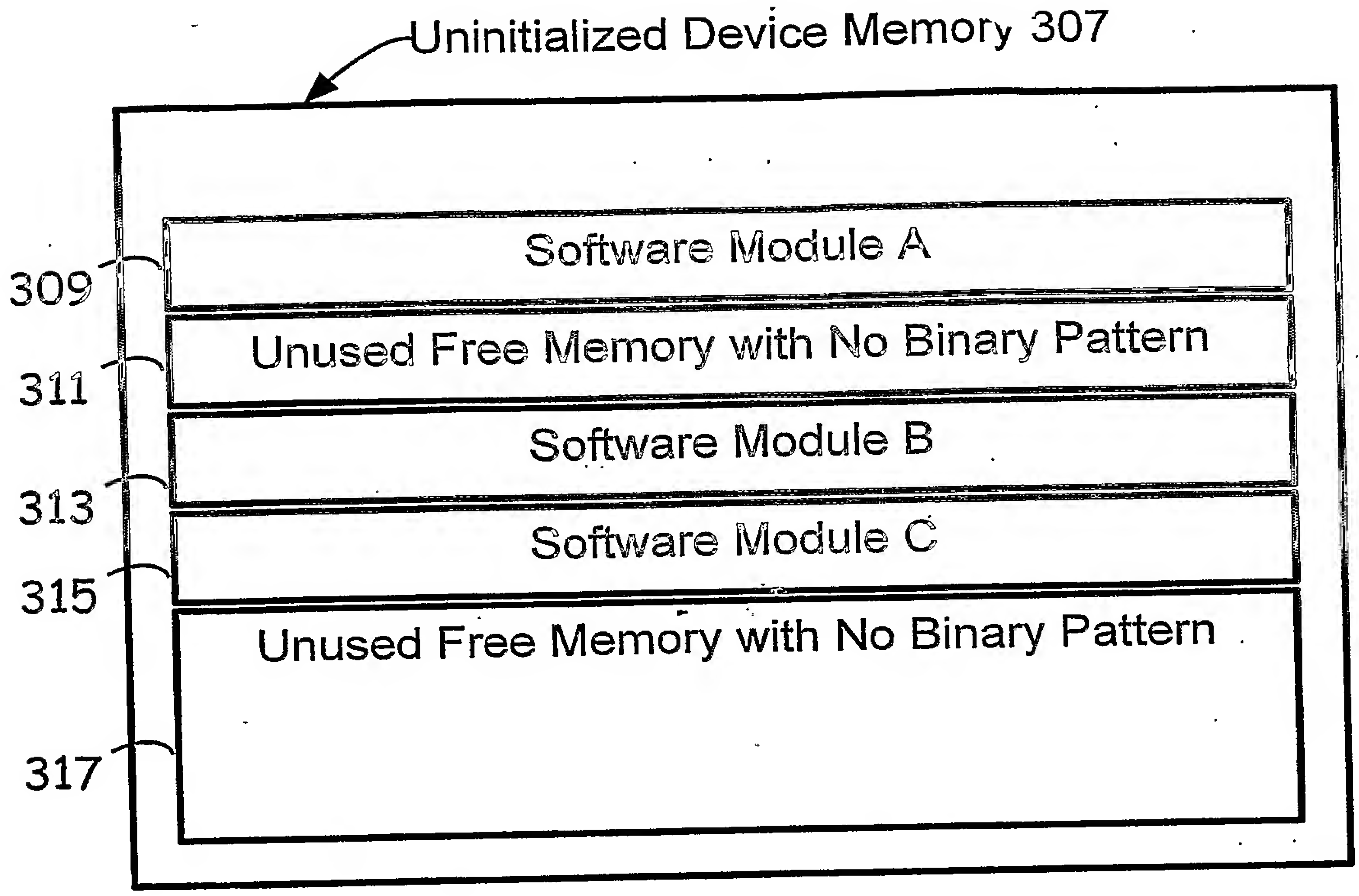

Uninitialized Device Memory 307
309
311
313
315
317
Software Module A
Unused Free Memory with No Binary Pattern
Software Module B
Software Module C
Unused Free Memory with No Binary Pattern

Fig. 3A

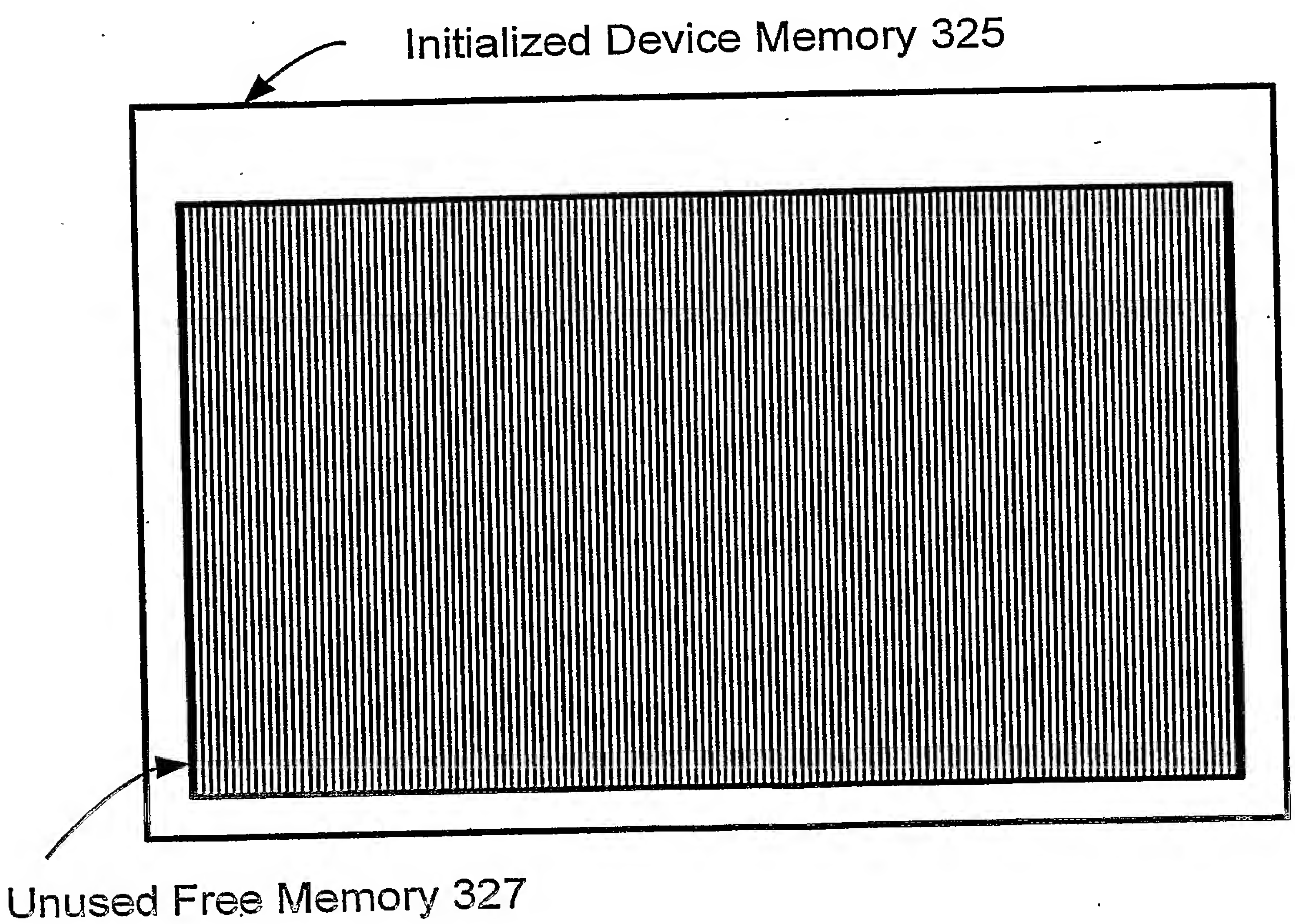

Initialized Device Memory 325
Unused Free Memory 327

Fig. 3B

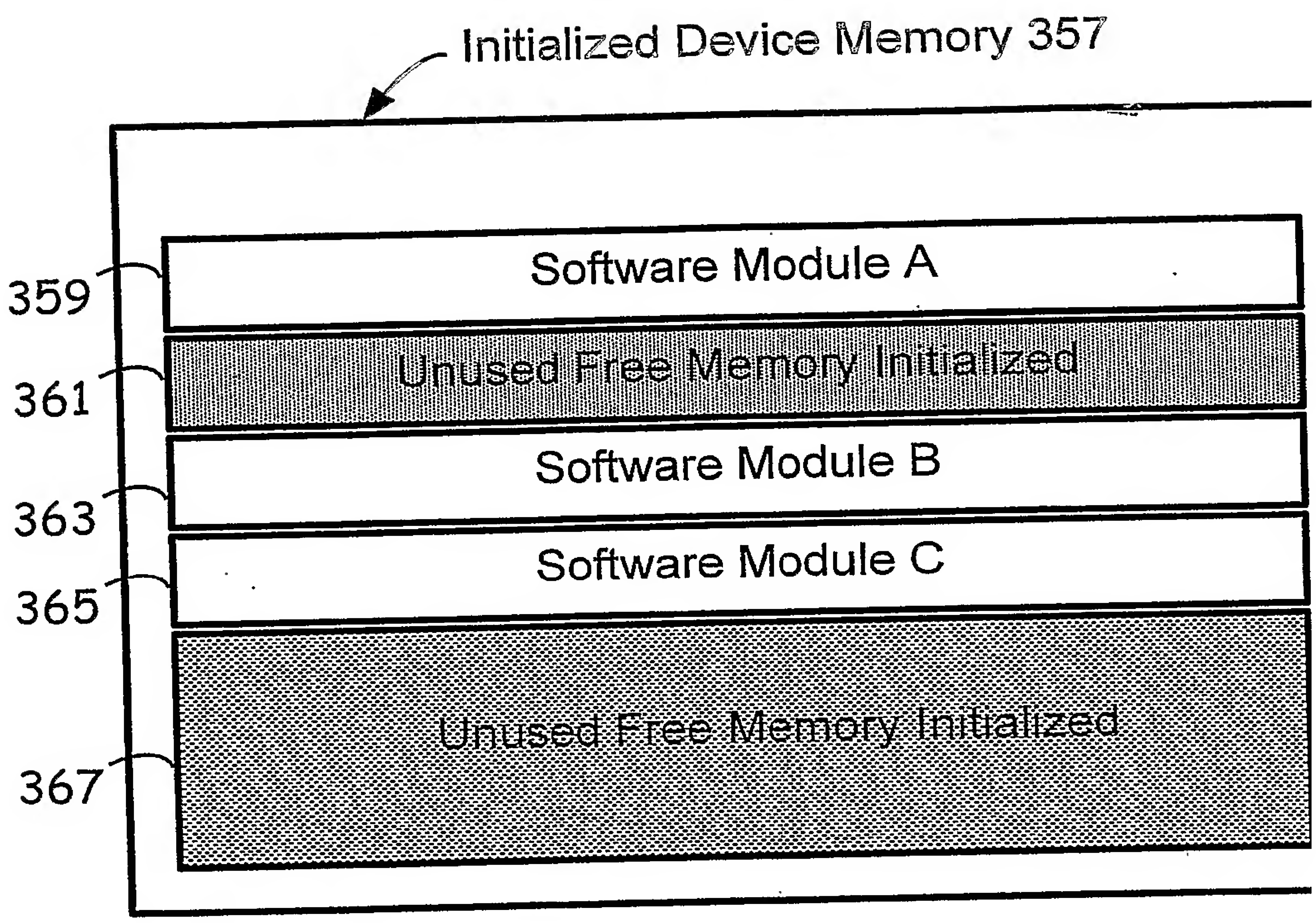

Initialized Device Memory 357
359
Software Module A
361
Unused Free Memory Initialized
363
Software Module B
365
Software Module C
367
Unused Free Memory Initialized

Fig. 3C

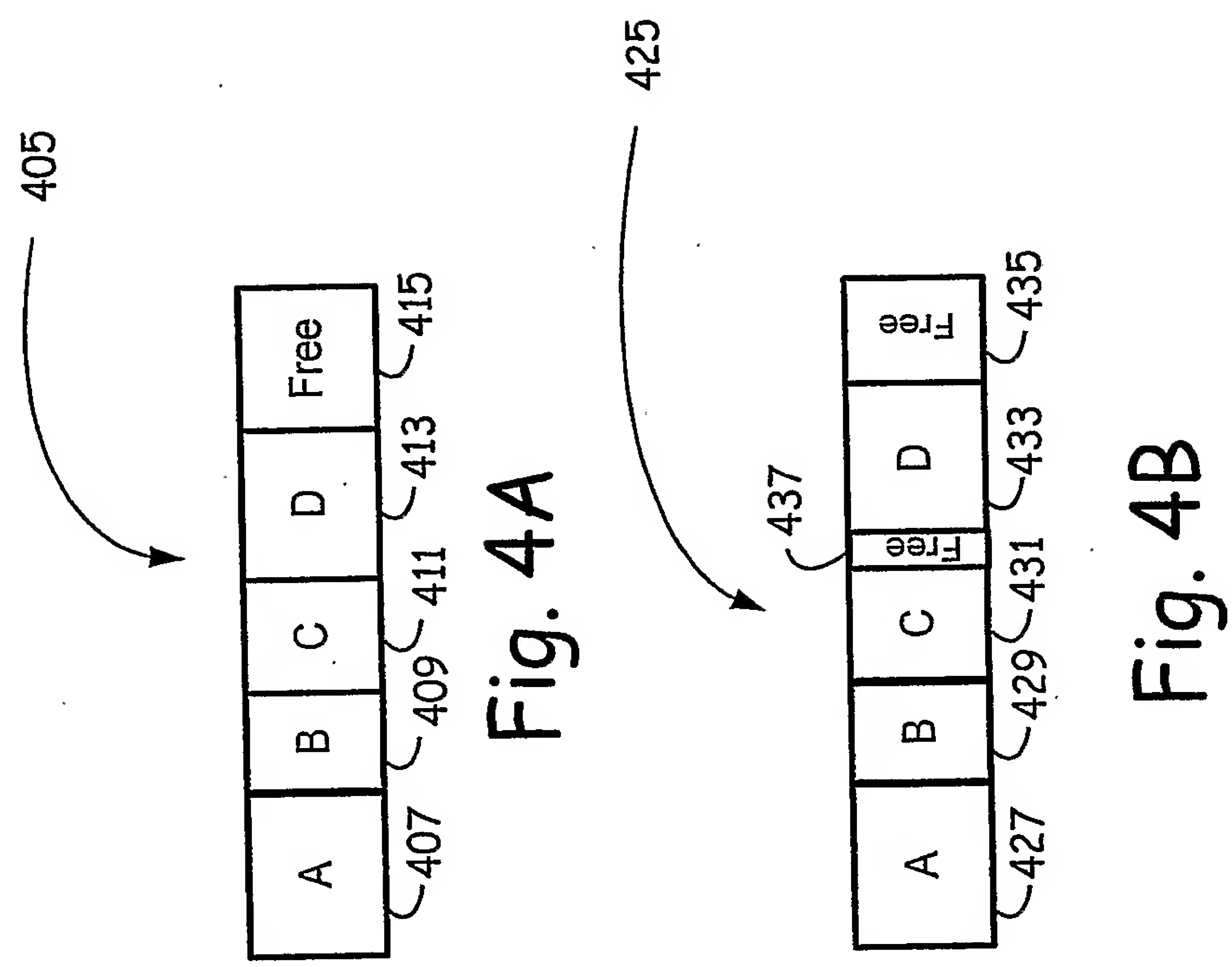

405
A
B
C
D
Free
407
409
411
413
415
425
A
B
C
Free
D
Free
427
429
431
437
433
435

Fig. 4A

Fig. 4B

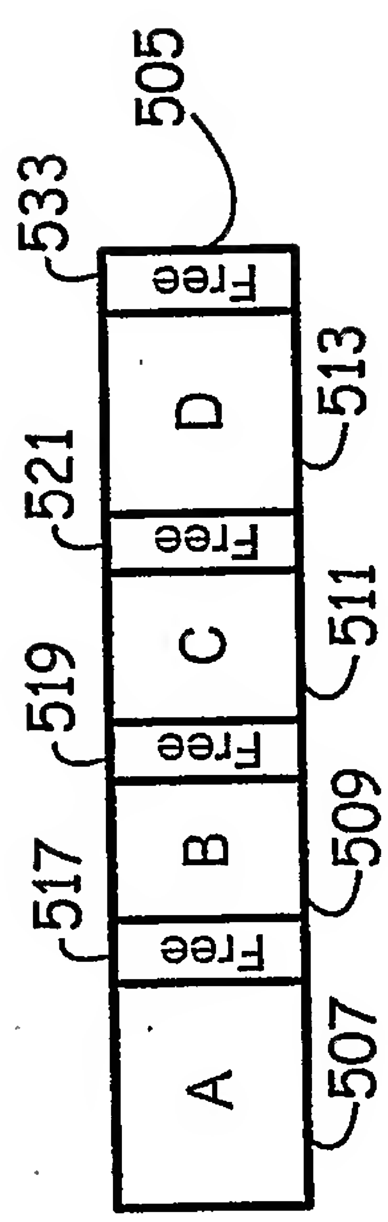

505
533
Free
D
513
521
Free
C
511
519
Free
B
509
517
Free
A
507

Fig. 5A

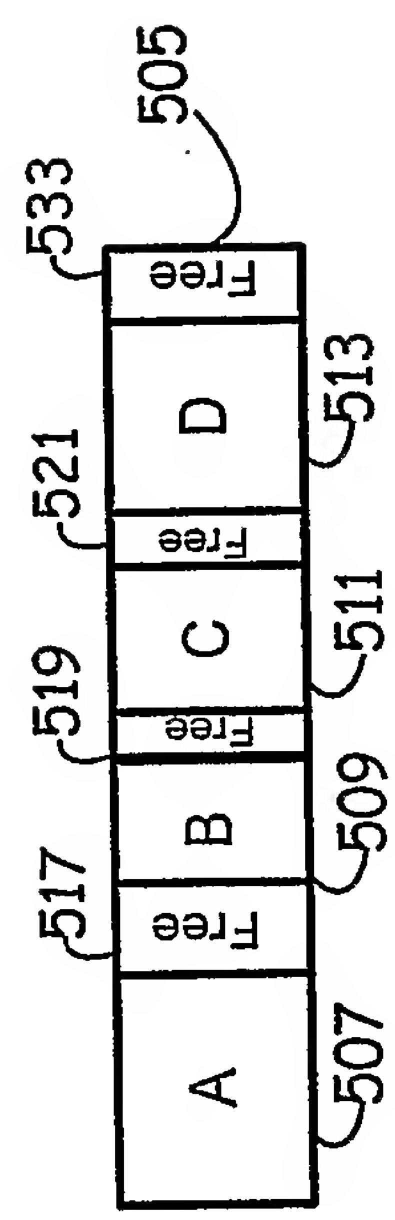

505
533
Free
D
513
521
Free
C
511
519
Free
B
509
517
Free
A
507

Fig. 5B

```
-o image.out
-m image.map

MEMORY {
  FLASH  (RX) : origin= 0x10000, length = 0x70000
  RAM   (RWXI): origin= 0x100000, length = 0x40000
}

SECTIONS
{
  .text   : load = FLASH
  .cinit  : load = FLASH
  .const  : load = FLASH
  .data   : load = FLASH
  .bss    : load = RAM
}
```

Fig. 6A

```
-o image.out
-m image.map

MEMORY {
  FLASH1 (RX) : origin= 0x10000, length = 0x10000
  FLASH2 (RX) : origin= 0x20000, length = 0x20000
  FLASH3 (RX) : origin= 0x40000, length = 0x20000
  FLASH4 (RX) : origin= 0x60000, length = 0x20000
  RAM   (RWXI): origin= 0x100000, length = 0x40000
}

SECTIONS
{

  .rom1   : load = FLASH2
  {
   C:\TIMAP\Gaps\img1\TI\Release\uadebug.obj (.text, .data, .const, .cinit
   C:\TIMAP\Gaps\img1\TI\Release\uadevice.obj (.text, .data, .const, .cinit
   C:\TIMAP\Gaps\img1\TI\Release\uadeviceasm.obj (.text, .data, .const, .
  }

  .rom2   : load = FLASH3
  {
   C:\TIMAP\Gaps\img1\TI\Release\uaflash.obj (.text, .data, .const, .cinit)
   C:\TIMAP\Gaps\img1\TI\Release\uainit.obj (.text, .data, .const, .cinit)
   C:\TIMAP\Gaps\img1\TI\Release\samson.obj (.text, .data, .const, .cinit)
   C:\TIMAP\Gaps\img1\TI\Release\link1.obj (.text, .data, .const, .cinit)
  }

  .rom3   : load = FLASH4
  {
   C:\TIMAP\Gaps\img1\TI\Release\uamemory.obj (.text, .data, .const, .ci
   C:\TIMAP\Gaps\img1\TI\Release\uaui.obj (.text, .data, .const, .cinit)
   C:\TIMAP\Gaps\img1\TI\Release\uabootmem.obj (.text, .data, .const, .
   C:\TIMAP\Gaps\img1\TI\Release\link2.obj (.text, .data, .const, .cinit)
   C:\TIMAP\Gaps\img1\TI\Release\uistring.obj (.text, .data, .const, .cinit)
  }

  .text   : load = FLASH1
  .cinit  : load = FLASH1
  .const  : load = FLASH1
  .data   : load = FLASH1
  .bss    : load = RAM
}
```

Fig. 6B

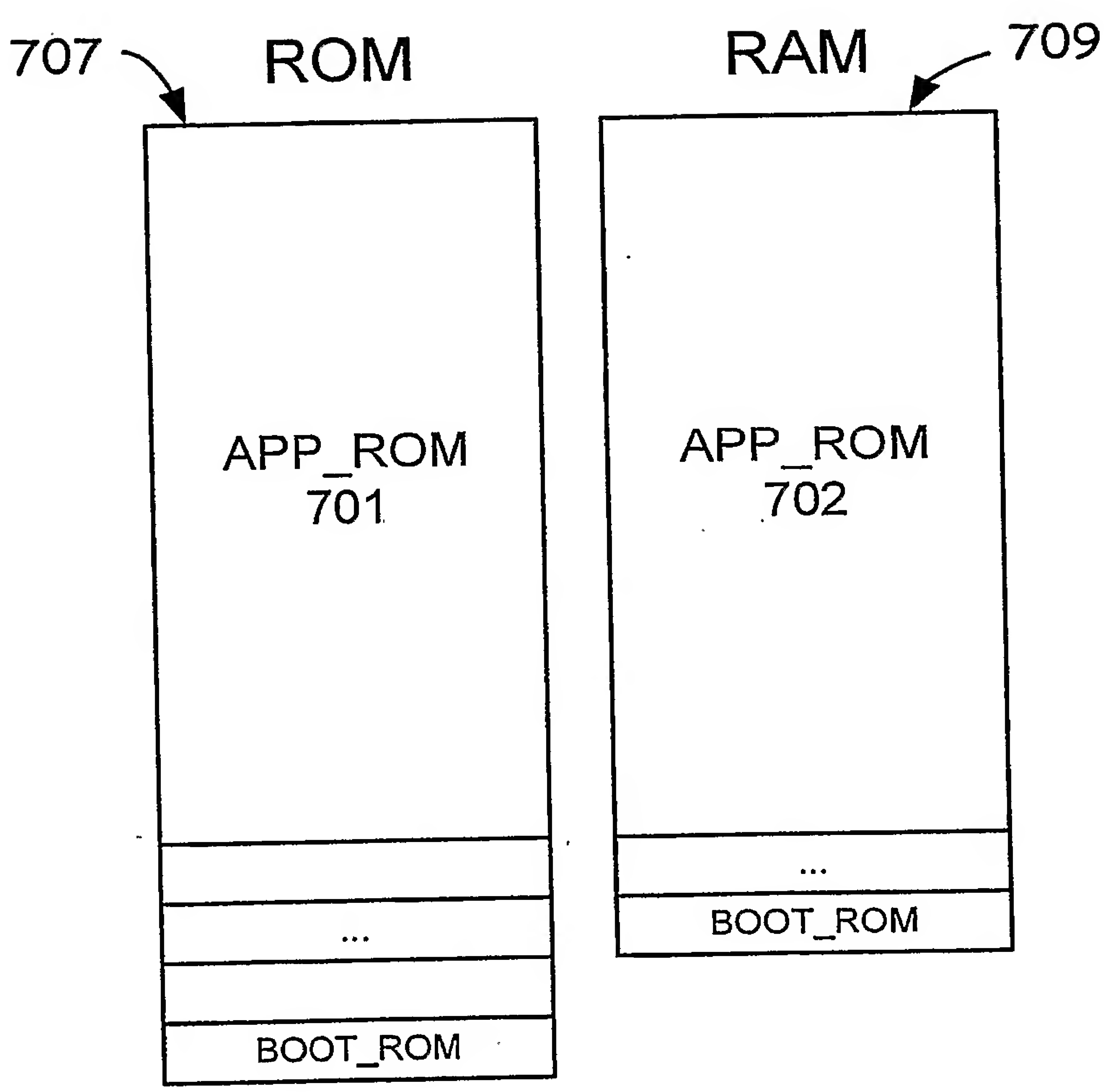


Figure 7A

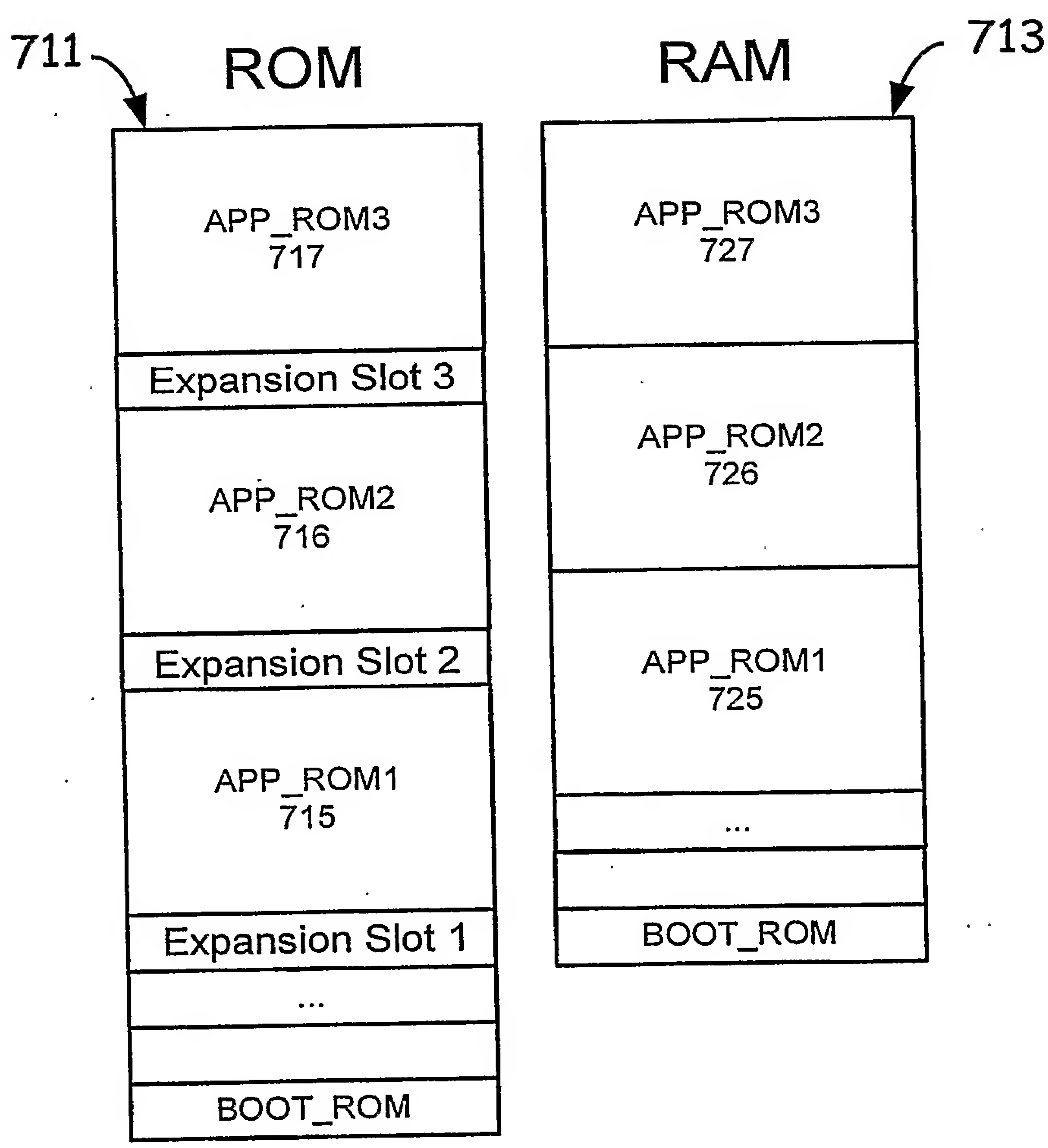

711
ROM
RAM
713
APP_ROM3
717
Expansion Slot 3
APP_ROM2
716
Expansion Slot 2
APP_ROM1
715
Expansion Slot 1
...
BOOT_ROM
APP_ROM3
727
APP_ROM2
726
APP_ROM1
725
...
BOOT_ROM

Figure 7B

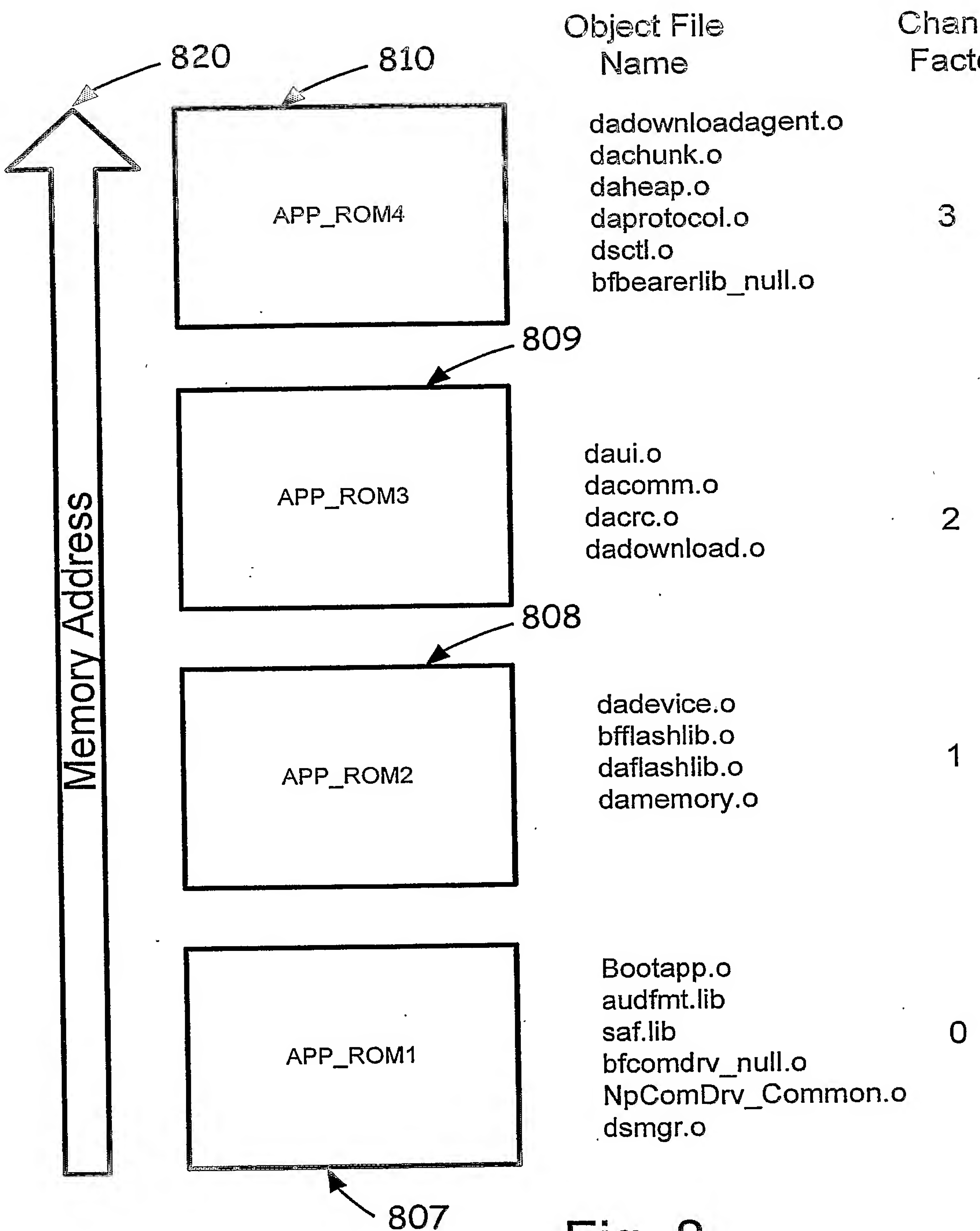

820
810
Object File
Name
Chan
Fact
APP_ROM4
dadownloadagent.o
dachunk.o
daheap.o
daprotocol.o
dsctl.o
bfbearerlib_null.o
3
809
APP_ROM3
daui.o
dacomm.o
dacrc.o
dadownload.o
2
808
APP_ROM2
dadevice.o
bfflashlib.o
daflashlib.o
damemory.o
1
APP_ROM1
Bootapp.o
audfmt.lib
saf.lib
bfcomdrv_null.o
NpComDrv_Common.o
dsmgr.o
0
Memory Address
807

Fig. 8

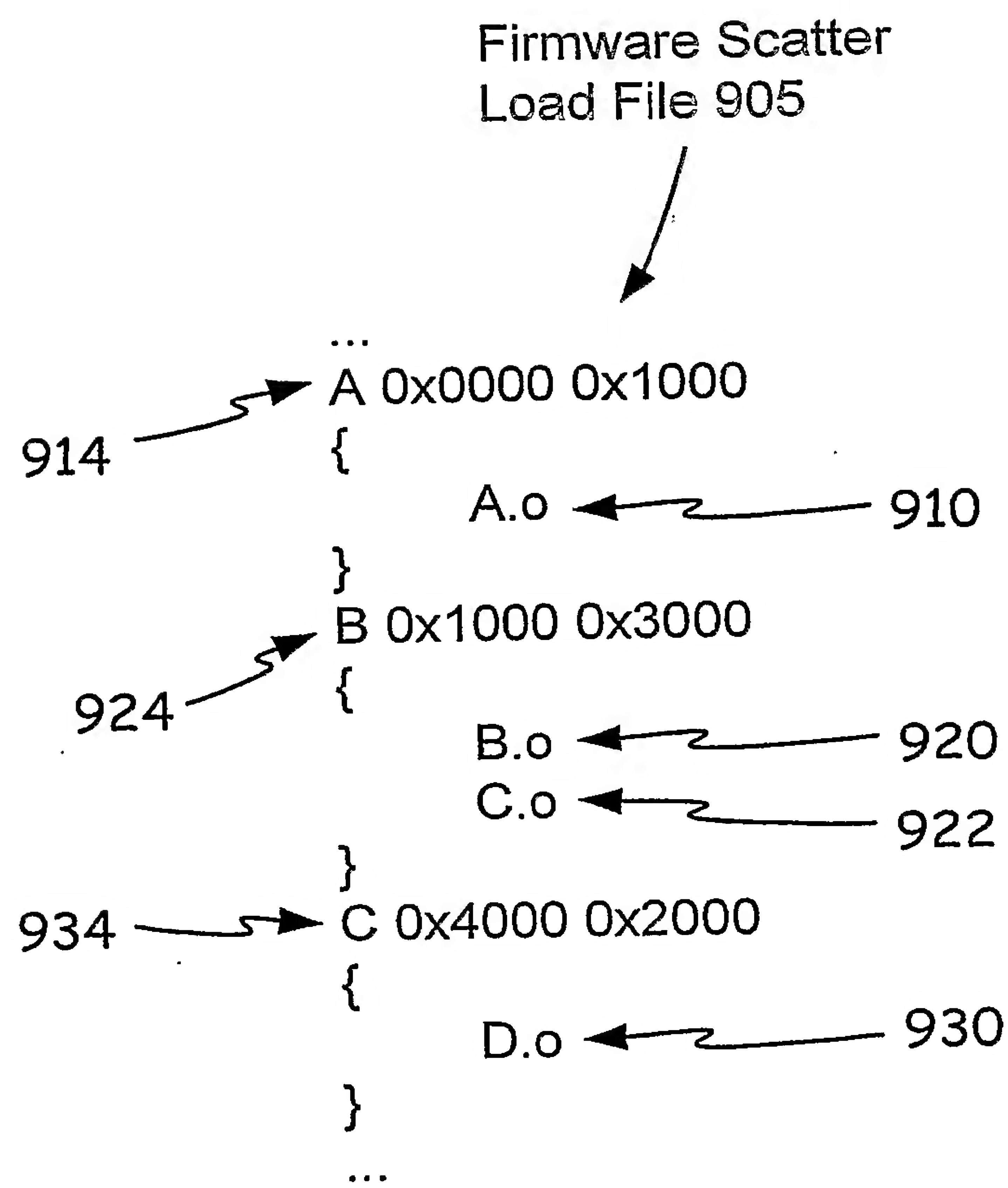

Firmware Scatter
Load File 905
...
A 0x0000 0x1000
{
A.o
}
B 0x1000 0x3000
{
B.o
C.o
}
C 0x4000 0x2000
{
D.o
}
...
914
910
924
920
922
934
930

Fig. 9

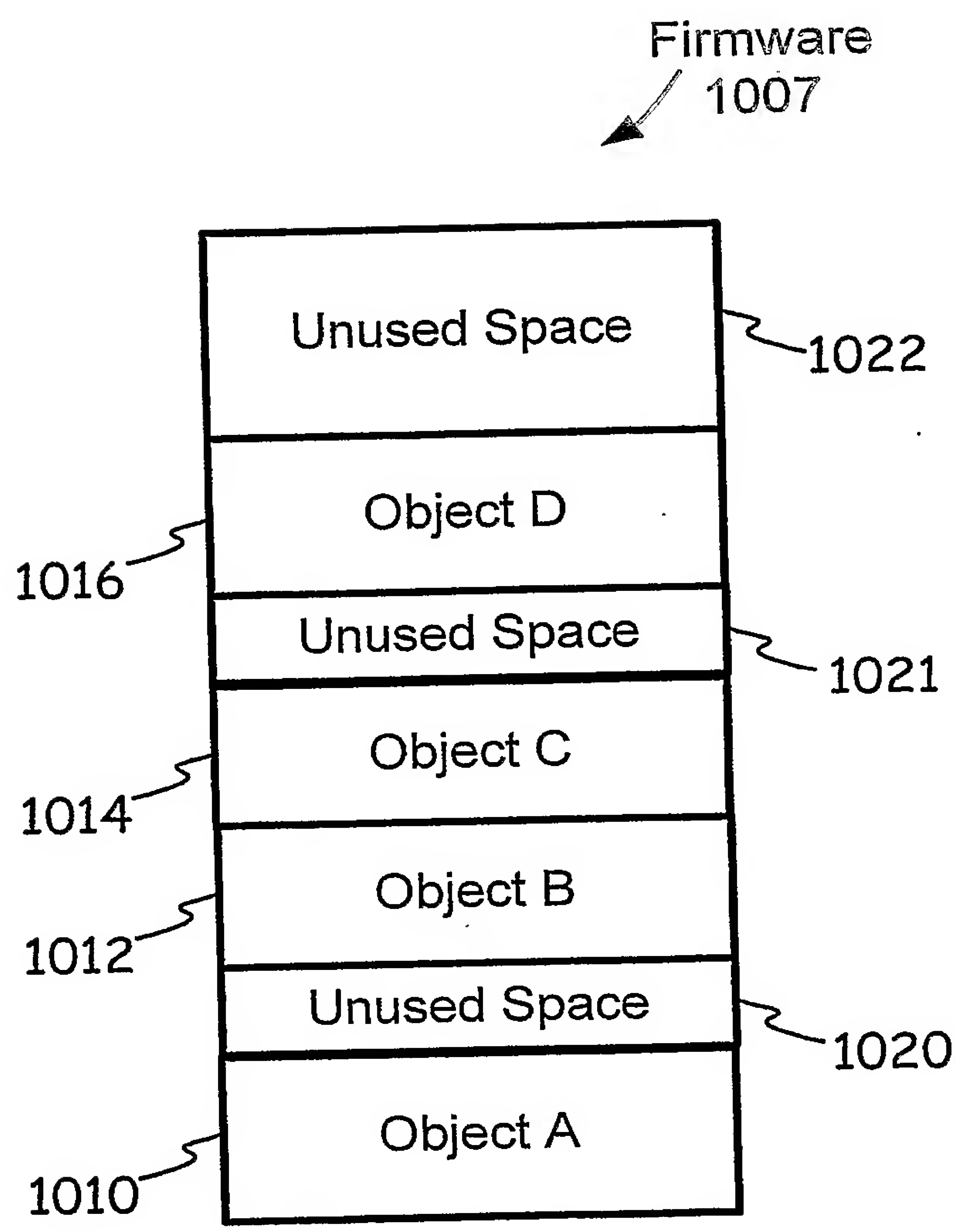

Firmware
1007
Unused Space
1022
Object D
1016
Unused Space
1021
Object C
1014
Object B
1012
Unused Space
1020
Object A
1010

Fig. 10